HOW to USE

The RoadRunner™

PORTABLE COMPUTER

WITH

Microsoft™ BASIC

MICRO OFFICE
SYSTEMS TECHNOLOGY, INC.

Microsoft(TM) BASIC Interpreter User's Guide for The RoadRunner Portable Computer

Document No. 8108A-530-02
Catalog No.
Part No.

Package Contents

Your BASIC package includes one BASIC cartridge which contains the BASIC.PGM file, and one manual containing the following documentation:

> BASIC User's Guide, which contains all the information about Microsoft BASIC that is specific to its use with *The Roadrunner*™ Portable Computer. This includes a description of the features affected by your implementation of Microsoft BASIC, information about cartridge file handling, and instructions on the use of assembly language subroutines.
>
> *Microsoft BASIC Reference Manual*, which contains descriptions of all Microsoft BASIC features. With the exceptions noted in the *User's Guide*, this information applies to all implementations of Microsoft BASIC.

Contents

Introduction

This implementation of BASIC utilized in the RoadRunner contains the features and commands of Microsoft BASIC as described below.

In 1975, Microsoft wrote the first BASIC interpreter for the microcomputer. Today Microsoft-BASIC is the BASIC you'll find on all the largest-selling microcomputers and in the application programs sold by many manufacturers and software houses. With over a million installations, Microsoft-BASIC is recognized as the industry standard.

Major Features of Microsoft BASIC

Four variable types: integer (-32768 through +32767), string (up to 255 characters), single precision floating-point (7 digits), double precision floating-point (16 digits)

Trace facilities (TRON/TROFF) for easier debugging

Error trapping using the ON ERROR GOTO statement

PEEK and POKE statements to read and write any memory location

Automatic line number generation and renumbering, including referenced line numbers

Arrays with multiple dimensions

Boolean operators OR, AND, NOT, XOR, EQV, and IMP

Formatted output with the complete PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion

Extensive program editing facilities with the EDIT command and edit mode subcommands

Assembly language subroutine calls: up to 10 per program with the USR function; limited only by available memory space with the CALL statement

IF/THEN/ELSE and nested IF/THEN/ELSE constructs

Variable length random access and sequential access cartridge files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, and MERGE

Note that features may vary from one implementation of MS-BASIC to the next. See Chapter 1 of this manual for an exact description of the features included in your implementation of Microsoft-BASIC with the RoadRunner.

System Requirements

Implementation of Microsoft-BASIC in the RoadRunner requires:

32K bytes of memory minimum:

24K for BASIC
Approximately 6K for the operating system
Additional memory to run programs

Files Included with Microsoft BASIC

Implementation of Microsoft BASIC in the RoadRunner includes the BASIC.PGM file. This is the version of Microsoft BASIC to be used on the RoadRunner.

How to Use this Manual

The BASIC User's Guide contains the information about Microsoft BASIC that is specific to its operation in the Roadrunner. Also provided are separate chapters on converting previously written programs to Microsoft BASIC, handling cartridge files, and calling assembly language subroutines. Briefly:

This Introduction tells you about Microsoft BASIC and its special features, and the conventions used in syntax notation. It also lists additional sources of information about programming in BASIC.

Chapter 1, "Microsoft BASIC with the CP/M Operating System," tells you how to initialize your implementation of Microsoft-BASIC and explains any differences between the features in your implementation and those described in the *Reference Manual*.

Chapter 2, "Converting Programs to Microsoft BASIC," describes the minor adjustments necessary when converting programs written in other BASICs to Microsoft-BASIC.

Chapter 3, "Cartridge File Handling," explains cartridge file handling procedures. This chapter can be read as an overview or used for reference if you encounter cartridge-related errors while running BASIC.

Chapter 4, "Microsoft BASIC Assembly Language Subroutines," provides information about calling assembly language subroutines.

Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate user-entered data. When the angle brackets enclose lowercase text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose uppercase text, the user must press the key named by the text; for example, <RETURN>.
{ }	Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
|	Vertical bars separate choices within braces. At least one of the entries separated by bars must be chosen unless the entries are also enclosed in square brackets.
...	Ellipses indicate that an entry can be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered, exactly as shown.

Learning More about BASIC

The manuals in this package provide complete reference information for your implementation of Microsoft BASIC. They do not, however, teach you how to write programs in BASIC. If you are new to BASIC or need help in learning to program, we suggest you read one of the following books:

Billings, Karen and Moursund, David. *Are You Computer Literate?* Beaverton, Oregon: Dilithium Press, 1979.

Albrecht, Robert L., Finkel, LeRoy, and Brown, Jerry. BASIC. New York: Wiley Interscience, 2nd ed., 1978.

Coan, James. *Basic BASIC*. Rochelle Park, N. J.: Hayden Book Company, 1978.

Dwyer, Thomas A. and Critchfield, Margot. *BASIC and the Personal Computer*. Reading, Mass.: Addison Wesley Publishing Company, 1978.

Simon, David E. *BASIC From the Ground Up*. Rochelle Park, N. J.: Hayden Book Company, 1978.

CHAPTER 1

MICROSOFT BASIC WITH THE CP/M COMPATIBLE OPERATING SYSTEM (I/OS)

This chapter tells you how to initialize BASIC in the RoadRunner.

1.1 HOW TO INITIALIZE MICROSOFT BASIC

To initialize BASIC *or* to run an application program:

a.) Insert BASIC cartridge into slot (A through D).

b.) Insert Data cartridge *or* an application program (if you have one), into an empty slot.

c.) Hit any key so the Roadrunner can identify the locations of the cartridges.

d.) Press the Function Key (F1 through F8) which corresponds to BASIC or to the application program, (if one has been inserted).

If BASIC is selected, the display will show:

```
MicroOffice RoadRunner                       BASIC Menu (BASIC o.nn)
Copyright 1977-1982 © by Microsoft
nnnnn Bytes free
Ok
RUN←  LIST  SAVE "  LOAD"  CONT←  LLIST  RESET←  BREAK
F1    F2    F3      F4     F5     F6     F7      F8
```

If an application program was selected, the application will load and run when the function key is pressed.

Once initialized, the BASIC cartridge can be removed.

NOTE

The function keys are pre-programmed to the commands displayed above.

1.2 HOW TO EXIT MICROSOFT BASIC

To exit BASIC and return to the Main menu, press EXIT key or type SYSTEM. This closes all files and then reloads operating system into memory without deleting any existing programs or data (i.e., performs a "warm start"). Note that a CMD-C, when executed within BASIC, returns to Microsoft BASIC command level, not to I/OS.

1.3 FILE NAMING CONVENTIONS

Filenames conform to standard naming conventions. All filenames can include A:, B:, C:, or D: as the first characters to specify a slot; otherwise, the currently selected slot is assumed. A default extension of .BAS is appended to LOAD, SAVE, MERGE, and RUN <filename> commands if no period (.) appears in the filename and the filename is less than nine characters long.

```
Examples    RUN "NEWFILE.EXE"
            RUN "A:NEWFILE.EXE"
            SAVE "NEWFILE"      (file is SAVEd with
                                .BAS extension)
```

1.4 MICROSOFT BASIC LANGUAGE DIFFERENCES FOR THE ROADRUNNER

Additional features are to be incorporated. Information pertaining to these features will be supplied as available.

1.4.1 FILES Command

This CP/M implementation of Microsoft BASIC for the RoadRunner includes a FILES command, which prints the names of files on the current cartridge. This command is not described in the *Reference Manual*.

Syntax	FILES[<filename>]
Purpose	To print the names of files residing on the current cartridge.
Remarks	If <filename> is omitted, all files on the cartridge in the currently selected slot will be listed. <filename> is a string formula which can contain question mark (?) "wild cards" to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

Examples	FILES	Lists all files on cartridge in currently selected slot
	FILES "*.BAS"	Lists all files with extension .BAS on cartridge in currently selected slot
	FILES "B:*.*"	Lists all files on cartridge in slot B.
	FILES "TEST?.BAS"	Lists all files with names that begin with TEST and end with .BAS on the currently selected slot

1.4.2 LOF Function

The LOF (length-of-file) function is available in this version of Microsoft-BASIC. This function is not described in the *Reference Manual*.

Syntax LOF(<file number>)

Action Returns the number of 128-byte records present in the last "extent" (128 records) read or written. If the file does not exceed one extent, LOF returns the true length of the file.

Example

```
110 IF NUM%>LOF(1) THEN PRINT
''INVALID ENTRY''
```

1.4.3 RESET Command

The RESET command is available in this version of Microsoft-BASIC and can be executed by pressing Function key F7 until F7 is changed.

Syntax RESET

Purpose To execute a cartridge system reset. After cartridges are exchanged in a slot, RESET must be executed before reading or writing to the new cartridge.

Remarks The RESET command also closes all open files on all cartridges and writes the updated directory information to the affected cartridges.

1.4.4 KEY $(X) Function

The key $(X) function is available in this version of Microsoft BASIC for the RoadRunner. It is used to access the current value of function key x.

Syntax KEY $(X) where X is a numerical expression which evaluates to 1, 2, 3, 4, 5, 6, 7, or 8.

Action A value outside this range will cause a "Function Call Error." This number indicates which function key you wish to examine, as they are numbered on the key tops. It will return a string from 0 to 10 characters in length, which is what would be entered if the user hit the function key himself.

Example You want to know (in your program) what Function Key 1 will enter ~~of~~ if it is typed. (We assume you have not modified the key yet.)

```
100 X$ = KEY $ (1)
```

will set the variable X$ to contain the string "LIST" since that is what is displayed when the user hits function key 1. KEY $(X) can also be used like any other string variable or function in formulas:

```
110 X$ = ''HI THERE! FUNCTION KEY 4 IS''
    +KEY$(4)+'', OKAY?''
120 PRINT ''FUNCTION KEY 3 IS''; KEY $(3)
```

1.4.5 KEY $(X) Statement

The KEY $(X) statement is available in this version of Microsoft BASIC. It is used to change the value of a function key.

Syntax — KEY $(X) = string-expression where x is a numerical expression which evaluates to 1, 2, 3, 4, 5, 6, 7, or 8.

Action — A value outside this range will cause a "Function Call Error." A 'String-expression' is any string formula or constant. Length may be from 0 to 10 characters; if it is longer than 10, the string will be truncated to 10 characters before it is used.

Function key X will be assigned the string-expression, and, until changed, will enter the string-expression when typed.

When MENU key on the RoadRunner is pressed, your changes will be shown.

Example — To change Function Key 2 to enter RUN without carriage return:

```
100 KEY$(2) = ''RUN''
```

To change Function Key 2 to enter RUN with carriage return:

```
110 KEY$(2) = ''RUN'' + CHR$(13)
```

To make a function key do nothing:

```
120 KEY$(5) = '' ''
```

Note: " " is the null string

If you try to put more than 10 characters into a key, i.e.:

```
130 KEY$(3) = ''01234567899876543210''
```

and then

```
PRINT KEY$(3)
```

you will get only 0123456789.

CHAPTER 2

CONVERTING PROGRAMS TO MICROSOFT BASIC

If you have programs written in a BASIC other than Microsoft BASIC, some minor adjustments may be necessary before they can be run with the Microsoft-BASIC Interpreter. This chapter lists some specific things to look for when converting BASIC programs.

2.1 STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Microsoft-BASIC statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for Microsoft-BASIC string concatenation.

In Microsoft-BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	*Microsoft BASIC*
`X$=A$(I)`	`X$=MID$(A$,I,1)`
`X$=A$(I,J)`	`X$=MID$(A$,I,J-I+1)`

If the substring reference is on the left side of the assignment operator and X$ is used to replace characters in A$, convert as follows:

Other BASIC	*Microsoft BASIC*
`A$(I)=X$`	`MID$(A$,I,1)=X$`
`A$(I,J)=X$`	`MID$(A$,I,J-I+1)=X$`

2.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form

```
10 LET B=C=0
```

to set B and C equal to zero. Microsoft-BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equalled 0. In Microsoft-BASIC, use a separate statement to define each variable. For example, convert the above statement to:

```
10 C=0:B=0
```

2.3 MULTIPLE STATEMENTS

Some BASICs use a backslash (\) to separate multiple statements on a line. With Microsoft-BASIC, be sure all statements on a line are separated by a colon (:).

2.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops.

CHAPTER 3
CARTRIDGE FILE HANDLING

Procedures for handling cartridge files are explained in this chapter. This information can also be useful if you are encountering cartridge-related errors while running Microsoft-BASIC.

3.1 FILENAMES

Wherever a filename is required in a cartridge command or statement, use a name that conforms to your operating system's requirements for filenames. As explained in Chapter 1, standard CP/M filenames can begin with A: or B: to specify the slot where a file is located; otherwise, the currently selected slot is assumed. In addition, the operating system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE, or LOAD command, if the filename contains no periods (.) and is less than 9 characters long.

Examples:	`B:NEWFILE.BAS`	
	`DATAFILE.DAT`	(current slot is assumed)
	`NEWFILE`	(current slot is assumed; .BAS extension will be appended with SAVE, RUN, MERGE, or LOAD commands)

3.2 PROGRAM FILE COMMANDS

The following commands are used to manipulate program files. When used with these commands, <filename> must be enclosed in quotation marks. Each of these commands is discussed in detail in Chapter 2 of the *Microsoft BASIC Reference Manual*.

`SAVE ''<filename>''[,A]`	Writes to the cartridge the program that currently resides in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.) (See "Protected Files," Section 3.3, for a discussion of how to SAVE protected files.)
`LOAD ''<filename>''[,R]`	Loads the program from cartridge into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and can access the same data files. (LOAD "<filename>",R and RUN "<filename>",R are equivalent.)

`RUN ''<filename>''[,R]`	Loads the program from cartridge into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN "<filename>",R and LOAD "<filename>",R are equivalent.)
`MERGE ''<filename>''`	Loads the program from cartridge into memory but does not delete the current contents of memory. The program line numbers on cartridge are merged with the line numbers in memory. If two lines have the same number, only the line from the cartridge program is saved. After a MERGE command has been executed, the "merged" program resides in memory, and BASIC returns to command level.
`KILL ''<filename>''`	Deletes the file from the cartridge. <filename> can be a program file, a sequential access data file, or a random access data file. The file extension must be included in the <filename>.
`NAME ''<oldfilename>'' AS ''<newfilename>''`	Changes the name of a cartridge file. NAME can be used with program files, random access files, or sequential access files. The file extension must be included in both filenames.

3.3 PROTECTED FILES

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command.

For example:

```
    SAVE ''MYPROG'',P
```

Warning Because a program saved this way cannot be listed or edited, you may also want to save an unprotected copy of the program for listing and editing.

3.4 CARTRIDGE DATA FILES—SEQUENTIAL AND RANDOM ACCESS

Two types of cartridge data files can be created and accessed by an Microsoft-BASIC program: sequential access files and random access files. Both are described in the following sections.

3.4.1 Sequential Access Files

Sequential access files are easier to create than are random access files, but they are limited in flexibility and speed when it comes to accessing data. Data is written to a sequential file as ASCII characters. These characters are stored, one after another (sequentially), in the order they are sent. They are read back the same way.

The statements and functions that are used with sequential files are:

```
OPEN
PRINT#
PRINT# USING
WRITE#
INPUT#
LINE INPUT#
EOF
LOC
CLOSE
```

See Chapters 2 and 3 of the *Microsoft BASIC Reference Manual* for a more detailed discussion of these commands.

3.4.1.1 Creating A Sequential Access File

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.	`OPEN ''O'',#1,''DATA''`
2. Write data to the file using the WRITE# statement. (You can use PRINT# instead, but consult Section 2.51 in the *Reference Manual* before doing so.)	`WRITE#1,A$;B$;C$`
3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.	`CLOSE #1` `OPEN ''I'',#1,''DATA''`
4. Use the INPUT# statement to read data from the sequential file into the program.	`INPUT#1,X$,Y$,Z$`

A program that creates a sequential file can also write formatted data to the cartridge with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING''####.##,'';A,B,C,D
```

could be used to write numeric data to cartridge without explicit delimiters. The comma at the end of the format string separates the items in the cartridge file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. For example,

```
    100 IF LOC(1)>50 THEN STOP
```

would end program execution if more than 50 sectors had been written to or read from file #1 since it was OPENed.

Program 1 creates a sequential file, named "DATA," from information you input at the terminal.

Program 1—Create a Sequential Data File

```
10 OPEN ''O'',#1,''DATA''
20 INPUT ''NAME'';N$
25 IF N$=''DONE'' THEN END
30 INPUT ''DEPARTMENT'';D$
40 INPUT ''DATE HIRED'';H$
50 PRINT#1,N$;'','';D$;'','';H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72
NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65
NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78
NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78
NAME? etc.
```

Program 2 accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978.

Program 2—Accessing a Sequential File

```
10 OPEN ''I'',#1,''DATA''
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)=''78'' THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15, which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

Then change line 40 to:

```
40 GOTO 15
```

3.4.1.2 Adding Data To A Sequential Access File

Data can be appended to an existing sequential access file. It is important, however, to follow carefully the procedure given below.

Warning If you have a sequential access file residing on cartridge and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential access file in "O" mode, you destroy its current contents.

The following procedure will add data to an existing file called "NAMES":

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY."
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY."
6. Rename "COPY" as "NAMES" and CLOSE it.
7. Now there is a file on cartridge called "NAMES" that includes all the previous data plus the data you just added.

Program 3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the cartridge file. Remember, LINE INPUT# will read in characters from the cartridge until it sees a carriage return (it does not stop at quotation marks or commas) or until it has read 255 characters.

Program 3—Adding Data to a Sequential File

```
  10 ON ERROR GOTO 2000
  20 OPEN ''I'',#1,''NAMES''
  30 REM IF FILE EXISTS, WRITE IT TO ''COPY''
  40 OPEN ''O'',#2,''COPY''
  50 IF EOF(1) THEN 90
  60 LINE INPUT#1,A$
  70 PRINT#2,A$
  80 GOTO 50
  90 CLOSE #1
 100 KILL ''NAMES''
 110 REM ADD NEW ENTRIES TO FILE
 120 INPUT ''NAME'';N$
 130 IF N$='' '' THEN 200 'CARRIAGE RETURN EXITS INPUT
     LOOP
 140 LINE INPUT ''ADDRESS? '';A$
 150 LINE INPUT ''BIRTHDAY? '';B$
 160 PRINT#2,N$
 170 PRINT#2,A$
 180 PRINT#2,B$
 190 PRINT:GOTO 120
 200 CLOSE
 205 REM CHANGE FILENAME BACK TO ''NAMES''
 210 NAME ''COPY'' AS ''NAMES''
2000 IF ERR=53 AND ERL=20 THEN OPEN ''O'',#2,''COPY'':
     RESUME 120
2010 ON ERROR GOTO 0
```

The error handling routine in line 2000 traps a "File not found" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

3.4.2 Random Access Files

Creating and accessing random access files requires more program steps than for sequential access files. However, there are advantages to using random access files, one of which is that random access files require less room on the cartridge, because Microsoft-BASIC stores them in packed binary format. (A sequential access file is stored as a series of ASCII characters.)

The biggest advantage of using random access files is that data can be accessed randomly, i.e., anywhere on the cartridge—it is not necessary to read through all the information, as with sequential access files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

CP/M versions 2.0 or later support large random access files. The maximum number of logical records is 32,767. If a record size of 256 is specified with the OPEN command (see Section 3.4.2.1), files up to 8 megabytes can be accessed.

The statements and functions that are used with random access files are:

```
OPEN
FIELD
LSET/RSET
GET
PUT
CLOSE
LOC
MKD$
MKI$
MKS$
CVD
CVI
CVS
```

See Chapters 2 and 3 of the *Reference Manual* for detailed discussion of these statements and functions.

3.4.2.1 Creating A Random Access File

The following program steps are required to create a random access file.

1. OPEN the file for random access (in "R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.	`OPEN ''R'',#1, ''FILE'',32`
2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random access file.	`FIELD #1,20 AS N$, 4 AS A$,8 AS P$`
3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.	`LSET N$=X$` `LSET A$=MKS$(AMT)` `LSET P$=TEL$`
4. Write the data from the buffer to the cartridge using the PUT statement.	`PUT #1,CODE%`

The LOC function with random files returns the "current record number." For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program 4 writes information that is input at the terminal to a random access file.

Program 4—Create a Random File

```
 10 OPEN "R",#1,"FILE",32
 20 FIELD #1,20 AS N$,4 AS A$,8 AS P$
 30 INPUT "2-DIGIT CODE";CODE%
 40 INPUT "NAME";X$
 50 INPUT "AMOUNT";AMT
 60 INPUT "PHONE";TEL$:PRINT
 70 LSET N$=X$
 80 LSET A$=MKS$(AMT)
 90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Warning Do not use a FIELDed string variable in an INPUT statement or in the left side of an assignment statement. This causes the pointer for that variable to point into string space instead of into the random access file buffer.

3.4.2.2 Accessing A Random Access File

The following program steps are required to access a random access file:

1. OPEN the file in "R" mode.

   ```
   OPEN ''R'',#1,
   ''FILE'',32
   ```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

   ```
   FIELD #1,20 AS N$,
   4 AS A$,8 AS P$
   ```

   Note In a program that performs both input and output on the same random access file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to the desired record into the random buffer.

   ```
   GET #1,CODE%
   ```

4. The data in the buffer may now be accessed by the program. Values that were made into strings when placed in the buffer (see "LSET," Section 3.4.2.1) must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

   ```
   PRINT N$
   PRINT CVS(A$)
   ```

Program 5 accesses the random access file “FILE” that was created in Program 4. When the two-digit code set up in Program 4 is input, the information associated with that code is read from the file and displayed.

Program 5—Access a Random File

```
10 OPEN ''R'',#1,''FILE'',32
20 FIELD #1,20 AS N$,4 AS A$,8 AS P$
30 INPUT ''2-DIGIT CODE'';CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING ''$$###.##'';CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

Program 6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number. It is assumed that the inventory will contain no more than 100 different part numbers.

Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 140-210 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

Program 6—Inventory

```
120 OPEN''R'',#1,''INVEN.DAT'',39
130 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
140 PRINT:PRINT ''FUNCTIONS:'':PRINT
150 PRINT 1,''INITIALIZE FILE''
160 PRINT 2,''CREATE A NEW ENTRY''
170 PRINT 3,''DISPLAY INVENTORY FOR ONE PART''
180 PRINT 4,''ADD TO STOCK''
190 PRINT 5,''SUBTRACT FROM STOCK''
200 PRINT 6,''DISPLAY ALL ITEMS BELOW REORDER LEVEL''
210 PRINT:PRINT:INPUT''FUNCTION'';FUNCTION
220 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
        ''BAD FUNCTION NUMBER'':GOTO 140
230 ON FUNCTION GOSUB 900,250,390,480,560,680
```

```
240 GOTO 210
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT''OVERWRITE'';A$:
    IF A$<>''Y'' THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT ''DESCRIPTION'';DESC$
300 LSET D$=DESC$
310 INPUT ''QUANTITY IN STOCK'';Q%
320 LSET Q$=MKI$(Q%)
330 INPUT ''REORDER LEVEL'';R%
340 LSET R$=MKI$(R%)
350 INPUT ''UNIT PRICE'';P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT ''NULL ENTRY'':RETURN
420 PRINT USING ''PART NUMBER ###'';PART%
430 PRINT D$
440 PRINT USING ''QUANTITY ON HAND #####'';CVI(Q$)
450 PRINT USING ''REORDER LEVEL #####'';CVI(R$)
460 PRINT USING ''UNIT PRICE $$##.##'';CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT ''NULL ENTRY'':RETURN
510 PRINT D$:INPUT ''QUANTITY TO ADD '';A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT ''NULL ENTRY'':RETURN
590 PRINT D$
600 INPUT ''QUANTITY TO SUBTRACT'';S%
610 Q%=CVI(Q$)
620 IF(Q%-S%)<0 THEN PRINT ''ONLY'';Q%;'' IN STOCK'':GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT ''QUANTITY NOW'';Q%;
        '' REORDER LEVEL'';CVI(R$)
```

```
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;''QUANTITY'';
                CVI(Q$)TAB(50)''REORDERLEVEL'';CVI(R$)
730 NEXT I
740 RETURN
840 INPUT ''PART NUMBER'';PART%
850 IF(PART%<1)OR(PART%>100)THENPRINT''BADPARTNUMBER'':
          GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT ''ARE YOU SURE'';B$:IF B$<>''Y'' THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

CHAPTER 4

MICROSOFT BASIC ASSEMBLY LANGUAGE SUBROUTINES

All versions of Microsoft BASIC provide interfacing with assembly language subroutines with the USR function and the CALL statement.

The USR function allows assembly language subroutines to be called in the same way BASIC intrinsic functions are called. (See Chapter 3 of the *Microsoft BASIC Reference Manual* for a discussion of intrinsic functions.)

The CALL statement, however, is the recommended method of interfacing assembly language subroutines. It is compatible with more languages than is the USR function call, it produces more readable source code, and it can pass multiple arguments. The calling sequence is the same as that for Microsoft(TM) FORTRAN, Microsoft(TM) COBOL, and Microsoft BASIC Compilers.

4.1 MEMORY ALLOCATION

Memory space should be set aside for an assembly language subroutine before it is loaded. To do so, use the /M: switch during initialization (see Chapter 1) to enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). Microsoft BASIC uses all memory available from its starting location upwards, so only the topmost locations in memory can be set aside for user subroutines.

It is important to avoid stack overflow when calling assembly language subroutines. Therefore, if additional stack space is needed when an assembly language subroutine is to be called, the Microsoft BASIC stack can be saved and a new stack set up for use by the assembly language subroutine. The Microsoft BASIC stack must be restored, however, before returning from the subroutine.

The assembly language subroutine can be loaded into memory by means of the operating system or the Microsoft-BASIC POKE statement. If the user has the Microsoft(TM) Macro Assembler, routines can be assembled with the Macro Assembler and loaded using the Microsoft(TM) LINK Linking Loader.

4.2 USR FUNCTION CALLS

The syntax of the USR function is:

`USR[<digit>](argument)`

where <digit> is any number from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR routine is being called and corresponds to the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A and its corresponding type of argument can be one of the following:

Value in A	*Type of Argument*
2	Two-byte integer (two's complement)
3	String
4	Single precision floating-point number
8	Double precision floating-point number

If the argument is a number, the HL register pair points to the Floating-point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC+0 contains the lower 8 bits of the argument.

FAC+1 contains the upper 8 bits of the argument.

If the argument is a single precision floating-point number:

FAC+0 contains the lowest 8 bits of mantissa.

FAC+1 contains the middle 8 bits of mantissa.

FAC+2 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

FAC+3 is the exponent minus 128. The binary point is to the left of the most significant bit of the mantissa.

If the argument is a double precision floating-point number:

FAC-4 through FAC-1 contain four more bytes of mantissa than the single precision argument (FAC-4 contains the lowest 8 bits).

If the argument is a string, the DE register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255 characters). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

Warning If the argument is a string literal in the program, the string descriptor will point to program text. To avoid altering or destroying your program this way, add + " " to the string literal in the program.

For example:

```
A$=''BASIC'' + ''  ''
```

This will copy the string literal into string space and prevent alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in HL as the value of the function, forcing the value returned by the function to be an integer. To execute MAKINT, use the following sequence to return from the subroutine:

```
   PUSH      H          ;save value to be returned
   LHLD     xxx         ;get address of MAKINT routine
   XTHL                 ;save return on stack and
                        ;get back HL
   RET                  ;return
```

Also, the argument of the function, regardless of its type, can be forced to an integer by calling the FRCINT routine to get the integer value of the argument in HL. To execute FRCINT, use the following routine:

```
   LXI     H,SUB1       ;get address of subroutine
                        ;continuation
   PUSH      H          ;place on stack
   LHLD     xxx         ;get address of FRCINT
   PCHL
SUB1: . . . . .
```

4.3 CALL STATEMENT

Assembly language subroutine calls can also be made with the CALL statement. The syntax is:

```
    CALL <variable name>[(<argument list>)]
```

where <variable name> contains an address that is the starting point in memory of the subroutine. <variable name> cannot be an array variable name. <argument list> contains the arguments that are passed to the external subroutine. <argument list> can contain only variables.

A Microsoft-BASIC CALL statement with no arguments generates a simple 8080 "CALL" instruction. The corresponding subroutine should return with a simple "RET." (CALL and RET are 8080 opcodes—see an 8080 microprocessor reference manual for details.)

A subroutine CALL with arguments results in a more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2. If the number of parameters is greater than 3, they are passed as follows:

   Parameter 1 in HL.

   Parameter 2 in DE.

   Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that with this scheme, the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or type of parameters.

If the subroutine expects more than 3 parameters and needs to transfer them to a local data area, a system subroutine named $AT will perform this transfer. The $AT routine is listed below (it is located in the Microsoft FORTRAN library, FORLIB.REL). The routine is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2).

The subroutine is responsible for saving the first two parameters before calling $AT. For example, if a subroutine expects 5 parameters, it should look like this:

```
SUBR:     SHLD      P1        ;SAVE PARAMETER 1
          XCHG
          SHLD      P2        ;SAVE PARAMETER 2
          MVI       A,3       ;NO. OF PARAMETERS LEFT
          LXI       H,P3      ;POINTER TO LOCAL AREA
          CALL      $AT       ;TRANSFER THE OTHER 3
                                       PARAMETERS
     .
     .
     .
          [Bodyofsubroutine]
     .
     .
     .
          RET                 ;RETURN TO CALLER
P1:       DS        2         ;SPACE FOR PARAMETER 1
P2:       DS        2         ;SPACE FOR PARAMETER 2
P3:       DS        6         ;SPACE FOR PARAMETERS 3-5
```

The argument transfer routine $AT is:

```
;       ARGUMENT TRANSFER
;BC     POINTS TO 3RD PARAM.
;HL     POINTS TO LOCAL STORAGE FOR PARAM 3
;A      CONTAINS THE # OF PARAMS TO XFER (TOTAL-2)


        ENTRY   $AT
$AT:    XCHG            ;SAVE HL IN DE
        MOV     H,B
        MOV     L,C     ;HL = PTR TO PARAMS
AT1:    MOV     C,M
        INX     H
        MOV     B,M
        INX     H       ;BC = PARAM ADR
        XCHG            ;HL POINTS TO LOCAL STORAGE
        MOV     M,C
        INX     H
        MOV     M,B
        INX     H       ;STORE PARAM IN LOCAL AREA
        XCHG            ;SINCE GOING BACK TO AT1
        DCR     A       ;TRANSFERRED ALL PARAMS?
        JNZ     AT1     ;NO, COPY MORE
        RET             ;YES, RETURN
```

When accessing parameters in a subroutine, remember that they are *pointers* to the actual arguments passed.

Note The programmer must match the *number*, *type*, and *length* of the arguments in the calling program with the parameters expected by the subroutine. This applies to Microsoft-BASIC subroutines, as well as those written in assembly language.

4.4 INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, register AL, and the PSW (Program Status Word). Interrupts should always be re-enabled before returning from the subroutine because an interrupt automatically disables all further interrupts once it is received. Under CP/M implementations of Microsoft-BASIC, all interrupt vectors are free.

INDEX

Microsoft(TM) BASIC

Reference Manual

Microsoft Corporation

Microsoft Corporation
Microsoft Building
10700 Northup Way
C-97200
Bellevue, Washington 98004

Document no. 8101-530-11
Catalog no.
Part no.

Contents

Introduction

Microsoft(TM) BASIC is the most extensive implementation of BASIC available for microprocessors. Microsoft BASIC meets the ANSI qualifications for BASIC, as set forth in document BSRX3.60-1978. Each release of Microsoft BASIC is compatible with previous versions.

How to Use this Manual

This manual is a reference for all implementations of Microsoft BASIC and for the Microsoft(TM) BASIC Compilers.

The manual is divided into three chapters plus three appendices. Chapter 1 covers a variety of topics, largely pertaining to data representation in Microsoft BASIC. Chapter 2 describes the syntax and semantics of every command and statement in Microsoft BASIC, ordered alphabetically. Chapter 3 describes all Microsoft BASIC intrinsic functions, also ordered alphabetically. The appendices contain a list of error messages and codes, a list of mathematical functions, and a list of ASCII character codes.

Additional information about programming Microsoft BASIC is covered in the *Microsoft BASIC User's Guide*. The *User's Guide* describes the features of Microsoft BASIC that are implemented for your machine. It also contains information relevant to your operating system and helpful hints about such matters as data I/O and assembly language subroutines.

Syntax Notation

Wherever the format for a statement or command is given, the following rules apply:

CAPS	Items in capital letters must be input as shown.
< >	Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.
[ ]	Items in square brackets ([ ]) are optional.

...	Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).
{ }	Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
|	Vertical bars separate the choices within braces. At least one of the entries separated by bars must be chosen unless the entries are also enclosed in square brackets.

All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.

LIST OF NEW COMMANDS ADDED TO MICROSOFT BASIC

Added KEY$(x) function and KEY$(x)= statement, used to access and redefine the function keys.

KEY$(x) as a function, to access the current value of function key x.

Format KEY$(x)
where x is a numerical expression which evaluates to 1,2,3,4,5,6,7, or 8. A value outside this range will cause a "Function Call Error." This number indicates which function key you wish to examine, as they are numbered on the function keys. It will return a string from 0 to 10 characters in length, which is what would be entered if the user hit the function key himself.

Example You want to know (in your program) what function key 1 will enter if it is typed. (We assume you have not modified the key yet.)

```
100 X$=KEY$(1)
```

will set the variable X$ to contain the string "LIST" since that is what is displayed when the user hits function key 1. KEY$(x) can also be used like any other string variable or function in formulas:

```
110 X$=''HI THERE! FUNCTION KEY 4 IS
    ''+KEY$(4)+'', OKAY?''
120 PRINT ''FUNCTION KEY 3 IS '';KEY$(3)
```

KEY$(x) as a statement, to change the value of a function key.

Format KEY$(x)=string_expression

where x is a numerical expression which evaluates to 1,2,3,4,5,6,7, or 8. A value outside this range will cause a "Function Call Error." 'string_expression' is any string formula or constant. Length may be from 0 to 10 characters; if it is longer than 10, the string will be truncated to 10 characters before it is used.

Function key x will be assigned the string–expression, and, until changed, function key x will enter when typed.

When you press the MENU key, your changes will be shown.

Example

To change function key 2 to enter RUN without typing a carriage return:

```
100 KEY$(2)=''RUN''
```

To change function key 2 to enter RUN with a carriage return:

```
110 KEY$(2)=''RUN''+CHR$(13)
```

To make a function key do nothing:

```
120 KEY$(5)='' ''    note: '' '' is the null
string
```

Here's what happens if you try to put more than 10 characters into a key:

```
130 KEY$(3)''01234567899876543210''
```

Now, if you PRINT KEY$(3), you will get only 0123456789.

Chapter 1 General Information about Microsoft BASIC

CHAPTER 1

GENERAL INFORMATION ABOUT MICROSOFT BASIC

1.1 INITIALIZATION

The procedure for initialization will vary with different implementations of Microsoft BASIC. Check the *Microsoft BASIC User's Guide* for the RoadRunner to determine how Microsoft BASIC is initialized with your operating system.

1.2 MODES OF OPERATION

When Microsoft BASIC is initialized, it displays the prompt "Ok". "Ok" indicates Microsoft BASIC is at command level; that is, it is ready to accept commands. At this point, Microsoft BASIC may be used in either of two modes: direct mode or indirect mode.

In direct mode, Microsoft BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. Direct mode is useful for debugging and for using Microsoft BASIC as a "calculator" for quick computations that do not require a complete program.

Indirect mode is used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 LINE FORMAT

Microsoft BASIC program lines have the following format (square brackets indicate optional input):

```
nnnnn BASIC statement[:BASIC statement...] <carriage return>
```

More than one BASIC statement may be placed on a line, but each must be separated from the last by a colon.

A Microsoft BASIC program line always begins with a line number and ends with a carriage return. A line may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by using the <line feed> key. <line feed> lets you continue typing a logical line on the next physical line without entering a <carriage return>.

1.3.1 Line Numbers

Every Microsoft BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory. Line numbers are also used as references in branching and editing. Line numbers must be in the range 0 to 65529.

A period (.) may be used in EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

1.4 CHARACTER SET

The Microsoft BASIC character set consists of alphabetic characters, numeric characters, and special characters.

The alphabetic characters in Microsoft BASIC are the upper case and lower case letters of the alphabet.

The Microsoft BASIC numeric characters are the digits 0 through 9.

In addition, the following special characters and terminal keys are recognized by Microsoft BASIC:

Character	Action
	Blank
=	Equals sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
↑	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point

[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
‘	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
—	Backslash or integer division symbol
@	At sign
	Underscore
<xC>	Deletes last character typed.
<SAVE>	Escapes edit mode subcommands. See Section 2.16.
<TAB>	Moves print position to next tab stop. Tab stops are set every eight columns.
<RETURN>	Moves to next physical line.
<carriage return>	Terminates input of a line.

1.4.1 Control Characters

Microsoft BASIC supports the following control characters:

Control Character (CMD key on RoadRunner)	Action
Control-A	Enters edit mode on the line being typed.
Control-C	Interrupts program execution and returns to BASIC command level.
Control-G	Rings the bell at the terminal.
Control-H	Backspaces. Deletes the last character typed.
Control-I	Tabs to the next tab stop. Tab stops are set every eight columns.

Control-O	Halts program output while execution continues. A second Control-O resumes output.
Control-R	Lists the line that is currently being typed.
Control-S	Suspends program execution.
Control-Q	Resumes program execution after a Control-S.
Control-U	Deletes the line that is currently being typed.

1.5 CONSTANTS

Constants are the values Microsoft BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks.

Examples:

```
''HELLO''
''$25,000.00''
''Number of Employees''
```

Numeric constants are positive or negative numbers. Microsoft BASIC numeric constants cannot contain commas. There are five types of numeric constants:

1. Integer constants	Whole numbers between -32768 and 32767. Integer constants do not contain decimal points.
2. Fixed-point	Positive or negative real numbers, constants i.e., numbers that contain decimal points.

3. Floating-point constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constants is 10-38 to 10+38.

Examples:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
```

(Double precision floating-point constants are denoted by the letter D instead of E. See Section 1.5.1.)

4. Hex constants

Hexadecimal numbers, denoted by the prefix &H.

Examples:

```
&H76
&H32F
```

5. Octal constants

Octal numbers, denoted by the prefix &O or &.

Examples:

```
&0347
&1234
```

1.5.1 Single And Double Precision Form For Numeric Constants

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with 7 digits of precision, and printed with up to 6 digits of precision. Double precision numeric constants are stored with 16 digits of precision and printed with up to 16 digits.

A single precision constant is any numeric constant that has one of the following characteristics:

1. Seven or fewer digits.

2. Exponential form using E.

3. A trailing exclamation point (!).

Examples:

```
46.8
-1.09E-06
3489.0
22.5!
```

A double precision constant is any numeric constant that has one of these characteristics:

1. Eight or more digits.

2. Exponential form using D.

3. A trailing number sign (#).

Examples:

```
345692811
-1.09432D-06
3489.0#
7654321.1234
```

1.6 VARIABLES

Variables are names used to represent values used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

1.6.1 Variable Names And Declaration Characters

Microsoft BASIC variable names may be any length. Up to 40 characters are significant. Variable names can contain letters, numbers, and the decimal point. However, the first character must be a letter. Special type declaration characters are also allowed—see below.

A variable name may not be a reserved word, but embedded reserved words are allowed. Reserved words include all Microsoft BASIC commands, statements, function names, and operator names. If a variable begins with FN, it is assumed to be a call to a user-defined function.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single precision, or double precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples of Microsoft BASIC variable names:

`PI#`	Declares a double precision value.
`MINIMUM!`	Declares a single precision value.
`LIMIT%`	Declares an integer value.
`N$`	Declares a string value.
`ABC`	Represents a single precision value.

There is a second method by which variable types may be declared. The Microsoft BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Section 2.12.

1.6.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32,767.

1.6.3 Space Requirements

The following table lists only the number of bytes occupied by the values represented by the variable names. Additional requirements may vary according to implementation.

Variables	Type	Bytes
	Integer	2
	Single Precision	4
	Double Precision	8

Arrays	Type	Bytes
	Integer	2 per element
	Single Precision	4 per element
	Double Precision	8 per element

Strings

3 bytes overhead plus the present contents of the string.

1.7 TYPE CONVERSION

When necessary, Microsoft BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

   Example:

   ```
   10 A%=23.42
   20 PRINT A%
   RUN
    23
   ```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

   Examples:

   ```
   10 D#=6#/7
   20 PRINT D#
   RUN
    .8571428571428571
   ```

   The arithmetic was performed in double precision and the result was returned in D# as a double precision value.

   ```
   10 D=6#/7
   20 PRINT D
   RUN
   .857143
   ```

   The arithmetic was performed in double precision and the result was returned to D (single precision variable), rounded, and printed as a single precision value.

3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

4. When a floating-point value is converted to an integer, the fractional portion is rounded.

   Example:

```
10 C%=55.88
20 PRINT C%
RUN
56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits (rounded) of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value.

   Example:

```
10 A=2.04
20 B#=A
30 PRINT A;B#
RUN
 2.04 2.039999961853027
```

1.8 EXPRESSIONS AND OPERATORS

An expression may be a string or numeric constant, a variable, or a combination of constants and variables with operators which produces a single value.

Operators perform mathematical or logical operations on values. The Microsoft BASIC operators may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

Each category is described in the following sections.

1.8.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	*Operation*	*Sample Expression*
↑	Exponentiation	X↑Y
-	Negation	-X
*,/	Multiplication, Floating-point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their Microsoft BASIC counterparts.

Algebraic Expression	BASIC Expression
X+2Y	X+Y*2
X-	X-Y/Z
	X*Y/Z
	(X+Y)/Z
(X	(X ↑ 2) ↑ Y
X	X ↑ (Y ↑ Z)
X(-Y)	X*(-Y) Two consecutive operators must be separated by parentheses.

1.8.1.1 Integer Division And Modulus Arithmetic -

Two additional operators are available in Microsoft BASIC: integer division and modulus arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.

Example:

```
        10\4=2          25.68\6.99=3
```

Integer division follows multiplication and floating-point division in order of precedence.

Modulus arithmetic is denoted by the operator MOD. Modulus arithmetic yields the integer value that is the remainder of an integer division.

Example:

```
        10.4 MOD 4=2 (10/4=2 with a remainder 2)
        25.68 MOD 6.99=5 (26/7=3 with a remainder 5)
```

Modulus arithmetic follows integer division in order of precedence.

1.8.1.2 Overflow And Division By Zero -

If, during the evaluation of an expression, division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation operator results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See "IF" statements, Section **2.26.**)

The relational operators are:

Operator	*Relation Tested*	*Example*
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See "LET," Section 2.30.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y<(T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z.

More examples:

```
IF SIN(X)<0 GOTO 1000         IF I MOD J<>0 THEN
K=K+1
```

1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in Table 1. The operators are listed in order of precedence.

Table 1. Microsoft BASIC Relational Operators Truth Table

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see "IF" statements, Section **2.26).**

Example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to 16-bit, signed, two's complement integers in the range -32768 to 32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63=binary 111111 and 16=binary 10000, so 63 AND 16=16.
15 AND 14=14	15=binary 1111 and 14=binary 1110, so 15 AND 14=14 (binary 1110).
-1 AND 8=8	-1=binary 1111111111111111 and 8=binary 1000, so -1 AND 8=8.

4 OR 2=6	4=binary 100 and 2=binary 10, so 4 OR 2=6 (binary 110).
10 OR 10=10	10=binary 1010, so 1010 OR 1010=1010 (decimal 10).
0-1 OR -2=-1	-1=binary 1111111111111111 and -2=binary 1111111111111110, so -1 OR -2=-1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. Microsoft BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All Microsoft BASIC intrinsic functions are described in Chapter 3.

Microsoft BASIC also allows "user-defined" functions that are written by the programmer. See "DEF FN," Section 2.11.

1.8.5 String Operations

Strings may be concatenated by using +.

Example:

```
10 A$=''FILE'' : B$=''NAME''
20 PRINT A$+B$
30 PRINT ''NEW ''+A$+B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=    < >    < >    <=    >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If during string comparison the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant.

Examples:

```
''AA''<''AB''
'FILENAME''=''FILENAME''
'X&''>''X#''
'CL ''>''CL''
'kg''>''KG''
'SMYTH''<''SMYTHE''
B$<''9/12/78''      where B$=''8/12/78''
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

1.9 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the <xC> key or with CMD-H. Rubout surrounds the deleted character(s) with backslashes. CMD-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type CMD-U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. Microsoft BASIC will automatically replace the old line with the new line.

More sophisticated editing capabilities are provided. See "EDIT," Section 2.16.

To delete the entire program currently residing in memory, enter the NEW command. (See Section 2.41.) NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

If an error causes program execution to terminate, an error message is printed. For a complete list of Microsoft BASIC error codes and error messages, see Appendix A.

Chapter 2 Microsoft BASIC Commands and Statements

CHAPTER 2

MICROSOFT BASIC COMMANDS AND STATEMENTS

Microsoft BASIC commands and statements are described in this chapter. Each description is formatted as follows:

Format	Shows the correct format for the instruction. See the "Introduction" to this manual for syntax notation.
Purpose	Tells what the instruction is used for.
Remarks	Describes in detail how the instruction is used.
Example	Shows sample programs or program segments that demonstrate the use of the instruction.
Note	Describes special cases or provides additional pertinent information.

2.1 AUTO

Format	AUTO [<line number>[,<increment>]]
Purpose	To generate a line number automatically after every carriage return.
Remarks	AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed. If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number. AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, Microsoft BASIC returns to command level.

Example

`AUTO 100,50`	Generates line numbers 100, 150, 200 ....
`AUTO`	Generates line numbers 10, 20, 30, 40 ....

2.2 CALL

Format CALL <variable name>[(<argument list>)]

Purpose To call an assembly language subroutine.

Remarks The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function, Section 3.41)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the external subroutine. <argument list> may contain only variables.

The CALL statement generates the same calling sequence used by Microsoft(TM) FORTRAN, Microsoft(TM) COBOL, and Microsoft(TM) BASIC Compilers.

Example

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
  .
  .
  .
```

Note In a Microsoft BASIC Compiler program, line 110 is not required because the address of MYROUT will be assigned by the linking loader at load time.

2.3 CHAIN

Format CHAIN [MERGE]<filename>[,[<line number exp>] [,ALL][,DELETE <range>]]

Purpose To call a program and pass variables to it from the current program.

Remarks <filename> is the name of the program that is called.

The COMMON statement may be used to pass variables (see Section 2.7).

Example 1

```
 10 REM THIS PROGRAM DEMONSTRATES CHAINING
    USING COMMON TO PASS VARIABLES.
 20 REM SAVE THIS MODULE ON DISK AS ''PROG1''
    USING THE A OPTION. 30 DIM A$(2),B$(2)
 40 COMMON A$(),B$()
 50 A$(1)=''VARIABLES IN COMMON MUST BE
    ASSIGNED''
 60 A$(2)=''VALUES BEFORE CHAINING.''
 70 B$(1)='''': B$(2) =''''
 80 CHAIN ''PROG2''
 90 PRINT: PRINT B$(1): PRINT: PRINT B$(2):
    PRINT
100 END
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.

Example 2

```
10 REM THE STATEMENT ''DIM A$(2),B$(2)'' MAY
   ONLY BE EXECUTED ONCE.
20 REM HENCE, IT DOES NOT APPEAR IN THIS
   MODULE.
30 REM SAVE THIS MODULE ON THE DISK AS
   ''PROG2'' USING THE A OPTION.
40 COMMON A$(),B$()
50 PRINT: PRINT A$(1);A$(2)
60 B$(1)=''NOTE HOW THE OPTION OF SPECIFYING
   A STARTING LINE NUMBER''
70 B$(2)=''WHEN CHAINING AVOIDS THE
   DIMENSION STATEMENT IN 'PROG1'.''
80 CHAIN ''PROG1'',90
90 END
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. See Section 2.7.

The MERGE option allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd.

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option.

Example 3

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING
   USING THE MERGE AND ALL OPTIONS.
20 REM SAVE THIS MODULE ON THE DISK AS
   ''MAINPRG''.
30 A$=''MAINPRG''
40 CHAIN MERGE ''OVRLAY1'',1010,ALL
50 END

1000 REM SAVE THIS MODULE ON THE DISK AS
     ''OVRLAY1'' USING THE A OPTION.
1010 PRINT A$; '' HAS CHAINED TO OVRLAY1.''
1020 A$=''OVRLAY1''
1030 B$=''OVRLAY2''
1040 CHAIN MERGE ''OVRLAY2'',1010,ALL,
     DELETE 1000-1050
1050 END

1000 REM SAVE THIS MODULE ON THE DISK AS
     ''OVRLAY2'' USING THE A OPTION.
1010 PRINT A$; '' HAS CHAINED TO '';B$:''.''
1020 END
```

The line numbers in <range> are affected by the RENUM command.

Note

The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

2.4 CLEAR

Format	CLEAR [,[<expression1>][,<expression2>]]
Purpose	To set all numeric variables to zero, all string variables to null, and to close all open files; and, optionally, to set the end of memory and the amount of stack space.
Remarks	<expression1> is a memory location which, if specified, sets the highest location available for use by Microsoft BASIC. <expression2> sets aside stack space for Microsoft BASIC. The default is 512 bytes or one-eighth of the available memory, whichever is smaller.
Note	Microsoft BASIC allocates string space dynamically. An "Out of string space" error occurs only if there is no free memory left for Microsoft BASIC to use.

2.5 CLOAD—Not used in Microsoft BASIC for the RoadRunner.

2.6 CLOSE

Format	CLOSE [[#]<file number>[,[#]<file number...>]]
Purpose	To conclude I/O to a cartridge file.
Remarks	<file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files. The association between a particular file and file number terminates upon execution of a CLOSE statement. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file. A CLOSE for a sequential output file writes the final buffer of output. The END statement and the NEW command always CLOSE all cartridge files automatically. (STOP does not close cartridge files.)
Example	See "Microsoft BASIC Cartridge I/O," in the Microsoft BASIC User's Guide.

2.7 COMMON

Format	COMMON <list of variables>
Purpose	To pass variables to a CHAINed program.
Remarks	The COMMON statment is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a proram, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example

```
100 COMMON A,B,C,D(),G$ 110 CHAIN ''PROG3'',
    10
  .
  .
  .
```

The current nonexecutable statements are:

```
COMMON
DEFDBL, DEFINT, DEFSNG, DEFSTR
DIM
OPTION BASE
REM
$INCLUDE
```

Array variables used in a COMMON statement must be declared in a preceding DIM statement.

The standard form of the COMMON statement is referred to as blank COMMON. Microsoft FORTRAN Compiler-style named COMMON areas are also supported; however, the variables are not preserved across CHAINs. The syntax for named COMMON is:

```
COMMON /<name>/ <list of variables>
```

where <name> consists of 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with Microsoft FORTRAN Compiler and assembly language routines without having to explicitly pass parameters in the CALL statement.

The blank COMMON size and order of variables must be the same in the CHAINing and CHAINed programs. With Microsoft BASIC Compiler, the best way to insure this is to place all blank COMMON declarations in a single include file and use the $INCLUDE statement in each program.

For example:

```
MENU.BAS
    10 $INCLUDE COMDEF
    .
    .
    . 1000 CHAIN ''PROG1''

PROG1.BAS
    10 $INCLUDE COMDEF
    .
    .
    . 2000 CHAIN ''MENU''

COMDEF.BAS
    100 DIM A(100),B$(200)
    110 COMMON I,J,K,A()
    120 COMMON A$,B$(),X,Y,Z
```

2.8 CONT

Format CONT

Purpose To continue program execution after a Control-C has been typed or a STOP or END statement has been executed.

Remarks Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error has occurred.

CONT is invalid if the program has been edited during the break.

Example See "STOP," Section 2.61.

2.9 CSAVE—Not used in Microsoft BASIC for the RoadRunner.

2.10 DATA

Format DATA <list of constants>

Purpose To store the numeric and string constants that are accessed by the program's READ statement(s). (See "READ," Section 2.54.)

Remarks DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas). Any number of DATA statements may be used in a program. READ statements access DATA statements in order (by line number). The data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format; i.e., fixed-point, floating-point, or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 2.57).

Example `See ''READ,'' Section 2.54.`

2.11 DEF FN

Format DEF FN<name>[(<parameter list>)]=<function definition>

Purpose To define and name a function that is written by the user.

Remarks <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function.

<parameter list> consists of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas.

<function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

This statement may define either numeric or string functions. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example

```
 .
 .
410 DEF FNAB(X,Y)=X↑3/Y↑2
420 T=FNAB(I,J)
 .
 .
```

Line 410 defines the function FNAB. The function is called in line 420.

2.12 DEFINT/SNG/DBL/STR

Format	DEF<type> <range(s) of letters> where <type> is INT, SNG, DBL, or STR
Purpose	To declare variable types as integer, single precision, double precision, or string.
Remarks	Any variable names beginning with the letter(s) specified in <range of letters> will be considered the type of variable specified in the <type> portion of the statement. However, a type declaration character always takes precedence over a DEFtype statement. (See "Variable Names and Declaration Characters," Section 1.6.1.) If no type declaration statements are encountered, Microsoft BASIC assumes all variables without declaration characters are single precision variables.

Examples

10 DEFDBL L-P	All variables beginning with the letters L, M, N, O, and P will be double precision variables.
10 DEFSTR A	All variables beginning with the letter A will be string variables.
10 DEFINT I-N,W-Z	All variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

2.13 DEF USR

Format DEF USR[<digit>]=<integer expression>

Purpose To specify the starting address of an assembly language subroutine.

Remarks <digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See "Assembly Language Subroutines," in the *Microsoft BASIC User's Guide.*

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example

```
.
.
.
200 DEF USR0=24000
210 X=USR0(Y↑2/2.89)
.
.
.
```

2.14 DELETE

Format DELETE [<line number>][-<line number>]

Purpose To delete program lines.

Remarks Microsoft BASIC always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples

DELETE 40	Deletes line 40.
DELETE 40-100	Deletes lines 40 through 100, inclusive.
DELETE -40	Deletes all lines up to and including line 40.

2.15 DIM

Format DIM <list of subscripted variables>

Purpose To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks If an array variable name is used without a DIM statement, the maximum value of the array's subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.46).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Theoretically, the maximum number of dimensions allowed in a DIM statement is 255. In reality, however, that number would be impossible, since the name and punctuation are also counted as spaces on the line, and the line itself has a limit of 255 characters. The number of dimensions is further limited by the amount of available memory.

Example

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
 .
 .
 .
```

2.16 EDIT

Format EDIT <line number>

Purpose To enter edit mode at the specified line.

Remarks In edit mode, it is possible to edit portions of a line without retyping the entire line. Upon entering edit mode, BASIC types the line number of the line to be edited, then it types a space and waits for an edit mode subcommand.

Edit Mode Subcommands

Edit mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. However, most of the edit mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When an integer is not specified, it is assumed to be 1.

Edit mode subcommands may be categorized according to the following functions:

1. Moving the cursor.
2. Inserting text.
3. Deleting text.
4. Finding text.
5. Replacing text.
6. Ending and restarting edit mode.
7. Entering edit mode from a syntax error.

Note In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ represents the Escape (or Altmode) key.

1. Moving the Cursor

Space bar	Use the space bar to move the cursor to the right. [i]Space bar moves the cursor i spaces to the right. Characters are printed as you space over them.
Rubout xC	In edit mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

I	I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, press Escape. If a <carriage return> is typed during an Insert command, the effect is the same as pressing Escape (SAVE) and then <carriage return>. During an Insert command, the Rubout, Delete, or Underscore key on the terminal may be used to delete characters to the left of the cursor. xC will print out the characters as you backspace over them. Delete and Underscore will print an Underscore for each character that you backspace over. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) sounds and the character is not printed.
X	The X subcommand extends the line. X moves the cursor to the end of the line, enters insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, press Escape (SAVE) or carriage return.

3. Deleting Text

D	[i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

H	H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for extending a line or replacing statements at the end of a line.

4. Finding Text

S	The subcommand [i]S<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor stops at the end of the line. All characters passed over during the search are printed.
K	The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C	The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by as many characters as are specified by i. After the ith new character is typed, change mode is exited and you will return to edit mode.

6. Ending and Restarting Edit Mode

<cr>	Typing a <carriage return> prints the remainder of the line, saves the changes you made, and exits edit mode.
E	The E subcommand has the same effect as <carriage return>, except the remainder of the line is not printed.
Q	The Q subcommand returns to Microsoft BASIC command level, *without* saving any of the changes that were made to the line in edit mode.

L	The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in edit mode. L is usually used to list the line when you first enter edit mode.
A	The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.
Control-A (CMD-A)	To enter edit mode on the line you are currently typing, type CMD-A. Microsoft BASIC responds with a <carriage return>, an exclamation point (!), and a space. The cursor will be positioned at the first character in the line. Proceed by typing an edit mode subcommand.

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT." will enter edit mode at the current line. (The line number symbol "." always refers to the current line.)

If an unrecognizable command or illegal character is input to Microsoft BASIC while in edit mode, BASIC sends a Control-G (bell) to the terminal, and the command or character is ignored.

7. Entering Edit Mode from a Syntax Error

When a syntax error is encountered during execution of a program, Microsoft BASIC automatically enters edit mode at the line that caused the error. For example:

```
10 K=2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and press <carriage return> (or the E subcommand), Microsoft BASIC reinserts the line. This causes all variable values to be lost. To preserve the variable values for examination, first exit edit mode with the Q subcommand. Microsoft BASIC will return to command level, and all variable values will be preserved.

2.17 END

Format END

Purpose To terminate program execution, close all files, and return to command level.

Remarks END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a "Break in line nnnnn" message to be printed. An END statement at the end of a program is optional. Microsoft BASIC always returns to command level after an END is executed.

Example

```
520 IF K>1000 THEN END ELSE GOTO 20
```

2.18 ERASE

Format ERASE <list of array variables>

Purpose To eliminate arrays from a program.

Remarks Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

Example

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

2.19 ERR AND ERL VARIABLES

When an error handling routine is entered, the variable ERR contains the error code for the error and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error handling routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test whether an error occurred in a direct statement, use IF 65535=ERL THEN .... Otherwise, use

```
IF ERR=error code THEN ...

IF ERL=line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered with RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. Microsoft BASIC error codes are listed in Appendix A.

2.20 ERROR

Format	ERROR <integer expression>
Purpose	To simulate the occurrence of a BASIC error, or to allow error codes to be defined by the user.
Remarks	The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix A), the ERROR statement will simulate the occurrence of that error and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by Microsoft BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to Microsoft BASIC.) This user-defined error code may then be conveniently handled in an error handling routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, Microsoft BASIC responds with the "Unprintable error" error message. Execution of an ERROR statement for which there is no error handling routine causes an error message to be printed and execution to halt.

Example 1

```
LIST
10 S=10
20 T=5
30 ERROR S+T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15              (You type this line.)
String too long       (BASIC types this line.)
Ok
```

Example 2

```
 .
 .
 .
110  ON ERROR GOTO 400 120 INPUT ''WHAT IS YOUR
   BET'';B 130 IF B>5000 THEN ERROR 210
 .
 .
400  IF ERR=210 THEN PRINT ''HOUSE LIMIT IS
     $5000''
410 IF ERL=130 THEN RESUME 120
 .
 .
 .
```

2.21 FIELD

Format FIELD [#]<file number>,<field width> AS <string variable>...

Purpose To allocate space for variables in a random file buffer.

Remarks Before a GET statement or PUT statement can be executed, a FIELD statement must be executed to format the random file buffer.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

```
FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See "LSET/RSET," Section 2.37, and "GET," Section 2.23.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128 bytes.)

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed will remain in effect at the same time.

Note *Do not use a FIELDed variable name in an INPUT or LET statement.* Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

Example 1

```
FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$
```

Allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See also "GET," Section 2.23, and "LSET/RSET," Section 2.37.)

Example 2

```
10 OPEN ''R,''#1,''A:PHONELST'',35
15 FIELD #1,2 AS RECNBR$,33 AS DUMMY$
20 FIELD #1,25 AS NAMES,10 AS PHONENBR$
25 GET #1
30 TOTAL=CVI(RECNBR)$
35 FOR I=2 TO TOTAL
40 GET #1, I
45 PRINT NAMES, PHONENBR$
50 NEXT I
```

Illustrates a multiple defined FIELD statement. In statement 15, the 35 byte field is defined for the first record to keep track of the number of records in the file. In the next loop of statements (35-50), statement 20 defines the field for individual names and phone numbers.

Example 3

```
10 FOR LOOP%=0 TO 7
20 FIELD #1,(LOOP%*16) AS OFFSETS,16 AS
   A$(LOOP%)
30 NEXT LOOP%
```

Shows the construction of a FIELD statement using an array of elements of equal size. The result is equivalent to the single declaration:

```
FIELD #1,16 AS A$(0),16 AS A$(1),...,16 AS A$
(6),16 AS A$(7)
```

Example 4

```
10 DIM SIZE% (NUMB%) : REM ARRAY OF FIELD SIZES
20 FOR LOOP%=0 TO NUMB%:READ SIZE%
(LOOP%) : NEXT LOOP%
30 DATA 9,10,12,21,41
  .
  .
  .
120  DIM A$(NUMB%) : REM ARRAY OF FIELDED
     VARIABLES
130  OFFSET%=0
140  FOR LOOP%=0 TO NUMB%
150  FIELD#1,OFFSET%ASOFFSET$,SIZE%(LOOP%) AS
     A$(LOOP%)
160  OFFSET%=OFFSET%+SIZE%(LOOP%)
170  NEXT LOOP%
```

Creates a field in the same manner as Example 3. However, the element size varies with each element. The equivalent declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$
(1),...SIZE%(NUMB%) AS A$(NUMB%)
```

2.22 FOR...NEXT

Format

```
FOR <variable>=x TO y [STEP z]
.
.
.
NEXT [<variable>][,<variable>...]
```

where x, y, and z are numeric expressions.

Purpose

To allow a series of instructions to be performed in a loop a given number of times.

Remarks

<variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is adjusted by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, Microsoft BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR... NEXT loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decreased each time through the loop. The loop is executed until the counter is less than the final value.

The counter must be an integer or single precision numeric constant. If a double precision numeric constant is used, a "Type mismatch" error will result.

The body of the loop is skipped if the initial value of the loop times the sign of the STEP exceeds the final value times the sign of the STEP.

Nested Loops

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within the context of another FOR... NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1 20
 3 30
 5 40
 7 50
 9 60
 Ok
```

Example 2

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

Note

Previous versions of Microsoft BASIC set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.

2.23 GET

Format	GET [#]<file number>[,<record number>]
Purpose	To read a record from a random cartridge file into a random buffer.
Remarks	<file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.
Example	See "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*
Note	After a GET statement has been executed, INPUT# and LINE INPUT# may be executed to read characters from the random file buffer.

2.24 GOSUB...RETURN

Format

```
GOSUB <line number>
.
.
.
RETURN
```

Purpose

To branch to and return from a subroutine.

Remarks

<line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program. A subroutine also may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause Microsoft BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Example

```
10 GOSUB 40
20 PRINT ''BACK FROM SUBROUTINE''
30 END
40 PRINT ''SUBROUTINE'';
50 PRINT '' IN'';
60 PRINT '' PROGRESS''
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

2.25 GOTO

Format GOTO <line number>

Purpose To branch unconditionally out of the normal program sequence to a specified line number.

Remarks If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example

```
LIST
10 READ R
20 PRINT ''R =''; R,
30 A=3.14*R↑2
40 PRINT ''AREA =''; A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5         AREA = 78.5
R = 7         AREA = 153.86
R = 12        AREA = 452.16
?Out of data in 10
Ok
```

2.26 IF...THEN[...ELSE] AND IF...GOTO

Format IF <expression> THEN {<statement(s)>|<line number>}

[ELSE {statement(s)>|<line number>}]

Format IF <expression> GOTO <line number>

[ELSE {<statement(s)>|<line number>}]

Purpose To make a decision regarding program flow based on the result returned by an expression.

Remarks If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example,

```
IF X>Y THEN PRINT ''GREATER'' ELSE IF Y>X
     THEN PRINT ''LESS THAN'' ELSE PRINT
     ''EQUAL''
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT ''A=C''
     ELSE PRINT ''A<>C''
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in direct mode, an "Undefined line" error results, unless a statement with the specified line number had previously been entered in indirect mode.

Note

When using IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1

```
200 IF I THEN GET#1,I
```

This statement GETs record number I if I is not zero.

Example 2

```
100 IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT ''OUT OF RANGE''
 .
 .
 .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the terminal.

2.27 INPUT

Format	INPUT[;] [<"prompt string">;]<list of variables>
Purpose	To allow input from the terminal during program execution.
Remarks	When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items or with the wrong type of value (numeric instead of string, etc.) causes the messsage "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Examples

```
10 INPUT X
20 PRINT X ''SQUARED IS'' X↑2
30 END
RUN
? 5         (The 5 was typed in by the user
             in response to the question mark.)
 5 SQUARED IS 25
Ok

LIST
10 PI=3.14
20 INPUT ''WHAT IS THE RADIUS'';R
30 A=PI*R↑2
40 PRINT ''THE AREA OF THE CIRCLE IS'';A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4 (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

2.28 INPUT#

Format INPUT#<file number>,<variable list>

Purpose To read data items from a sequential cartridge file and assign them to program variables.

Remarks <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed, or comma.

If Microsoft BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return, or line feed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Example See "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*

2.29 KILL

Format KILL <filename>

Purpose To delete a file from cartridge.

Remarks If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of cartridge files: program files, random data files, and sequential data files.

Example

```
200 KILL ''DATA1.DAT''
```

See also "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*

2.30 LET

Format [LET]<variable>=<expression>

Purpose To assign the value of an expression to a variable.

Remarks Notice the word LET is optional; i.e., the equal sign is sufficient for assigning an expression to a variable name.

Example

```
110 LET D=12
120 LET E=12↑2
130 LET F=12↑4
140 LET SUM=D+E+F
  .
  .
  .
```

or

```
110 D=12
120 E=12↑2
130 F=12↑4
140 SUM=D+E+F
  .
  .
  .
```

2.31 LINE INPUT

Format LINE INPUT[;] [<"prompt string">;]<string variable>

Purpose To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks <"prompt string"> is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of <"prompt string">. All input from the end of <"prompt string"> to the carriage return is assigned to <string variable>. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the line feed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT statement may be aborted by typing Control-C. Microsoft BASIC will return to command level and type "Ok". Typing CONT resumes execution at the LINE INPUT.

Example See "LINE INPUT#," Section 2.32.

2.32 LINE INPUT#

Format LINE INPUT#<file number>,<string variable>

Purpose To read an entire line (up to 254 characters), without delimiters, from a sequential cartridge data file to a string variable.

Remarks <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence. The next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a Microsoft BASIC program saved in ASCII format is being read as data by another program. (See "SAVE," Section 2.60.)

Example

```
10 OPEN ''O'',1,''LIST''
20 LINE INPUT ''CUSTOMER INFORMATION? '';C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN ''I'',1,''LIST''
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES 234,4
MEMPHIS LINDA JONES 234,4 MEMPHIS
Ok
```

2.33 LIST

Format 1 LIST[<line number>]

Format 2 LIST [<line number>][-[<line number>]]

Purpose To list all or part of the program currently in memory at the terminal.

Remarks Microsoft BASIC always returns to command level after a LIST is executed.

Format 1

If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either when the end of the program is reached or by typing CMD-C.) If <line number> is included, only the specified line will be listed.

Format 2

This format allows the following options:

1. If only the first <line number> is specified, that line and all higher-numbered lines are listed.

2. If only the second <line number> (i.e., [-[<line number>]]) is specified, all lines from the beginning of the program through that line are listed.

3. If both <line number(s)> are specified, the entire range is listed.

Examples	Format 1	
LIST	Lists the program currently in memory.	
	LIST 500	Lists line 500.
	Format 2	
	LIST 150-	Lists all lines from 150 to the end.
	LIST -1000	Lists all lines from the lowest number through 1000.
	LIST 150-1000	Lists lines 150 through 1000, inclusive.

2.34 LLIST

Format	LLIST [<line number>[-[<line number>]]]
Purpose	To list all or part of the program currently in memory at the line printer.
Remarks	LLIST assumes a 132-character-wide printer.
	Microsoft BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Format 2.
Note	LLIST and LPRINT are not included in all implementations of Microsoft BASIC.
Example	See the examples for "LIST," Format 2.

2.35 LOAD

Format LOAD <filename>[,R]

Purpose To load a file from cartridge into memory.

Remarks <filename> is the name that was used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide*, for information about possible filename extensions in your operating system.)

The R option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the R option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the R option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their cartridge data files.

Example

```
LOAD ''STRTRK'',R

LOAD ''B:MYPROG''
```

2.36 LPRINT AND LPRINT USING

Format	LPRINT [<list of expressions>]
	LPRINT USING <string exp>;<list of expressions>
Purpose	To print data at the line printer.
Remarks	Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.49 and Section 2.50.
	LPRINT assumes a 132-character-wide printer.
Note	LPRINT and LLIST are not included in all implementations of Microsoft BASIC.

2.37 LSET AND RSET

Format LSET <string variable>=<string expression>
RSET <string variable>=<string expression>

Purpose To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See "MKI$, MKS$, MKD$," Section 3.26.

Examples 150 LSET A$=MKS$(AMT)
160 LSET D$=DESC($)

See also "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*

Note LSET or RSET may also be used with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

2.38 MERGE

Format	MERGE <filename>
Purpose	To merge a specified cartridge file into the program currently in memory.
Remarks	<filename> is the name used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide*, for information about possible filename extensions under your operating system.) The file must have been SAVEd in ASCII format. (If not, a "Bad file mode" error occurs.) If any lines in the cartridge file have the same line numbers as lines in the program in memory, the lines from the file on cartridge will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on cartridge into the program in memory.) Microsoft BASIC always returns to command level after executing a MERGE command.
Example	`MERGE ''NUMBRS''`

2.39 MID$

Format MID$(<string exp1>,n[,m])=<string exp2>

where n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

Purpose To replace a portion of one string with another string.

Remarks The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional "m" refers to the number of characters from <string exp2> that will be used in the replacement. If "m" is omitted, all of <string exp2> is used. However, regardless of whether "m" is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

Example

```
10 A$=''KANSAS CITY, MO''
20 MID$(A$,14)=''KS''
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ is also a function that returns a substring of a given string. See Section 3.25.

2.40 NAME

Format NAME <old filename> AS <new filename>

Purpose To change the name of a cartridge file.

Remarks <old filename> must exist and <new filename> must not exist; otherwise, an error will result. After a NAME command, the file exists on the same cartridge, in the same area of cartridge space, with the new name.

Example

```
Ok
NAME ''ACCTS'' AS ''LEDGER''
Ok
```

In this example, the file that was formerly named ACCTS will now be named LEDGER.

2.41 NEW

Format NEW

Purpose To delete the program currently in memory and clear all variables.

Remarks NEW is entered at command level to clear memory before entering a new program. Microsoft BASIC always returns to command level after a NEW is executed.

Example

```
NEW
```

2.42 NULL—Not used

Format NULL <integer expression>

Purpose To set the number of nulls to be printed at the end of each line.

Remarks For 10 character-per-second tape punches, <integer expression> should be >=3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletype(R) and Teletype-compatible terminal screens. <integer expression> should be 2 or 3 for 30 CPS hard copy printers. The default value is 0.

Example

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
 .
 .
 .
```

Two null characters will be printed after each line.

2.43 ON ERROR GOTO

Format ON ERROR GOTO <line number>

Purpose To enable error handling and specify the first line of the error handling routine.

Remarks Once error handling has been enabled, all errors detected, including direct mode errors (e.g., syntax errors), will cause a jump to the specified error handling routine. If <line number> does not exist, an "Undefined line" error results.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error handling routine causes Microsoft BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

Note If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

Example

```
10 ON ERROR GOTO 1000
```

2.44 ON...GOSUB AND ON...GOTO

Format ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Purpose To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a noninteger, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), Microsoft BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example

```
100 ON L-1 GOTO 150,300,320,390
```

2.45 OPEN

Format OPEN <mode>,[#]<file number>,<filename>[,<reclen>]

Purpose To allow I/O to a cartridge file.

Remarks A cartridge file must be OPENed before any cartridge I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

O	Specifies sequential output mode.
I	Specifies sequential input mode.
R	Specifies random input/output mode.

<file number> is an integer expression whose value is between 1 and 15. The number is then associated with the file for as long as it is OPEN and is used to refer other other I/O statements to the file.

<filename> is a string expression containing a name that conforms to your operating system's rules for cartridge filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 128 bytes.

Note A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Example
```
10 OPEN ''I'',2,''INVEN''
```

See also "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*

2.46 OPTION BASE

Format OPTION BASE n

where n is 1 or 0

Purpose To declare the minimum value for array subscripts.

Remarks The default base is 0. If the statement

OPTION BASE 1

is executed, the lowest value an array subscript may have is 1.

Example

```
OPTION BASE 1
```

2.47 OUT

Format OUT I,J

where I and J are integer expressions in the range 0 to 255.

Purpose To send a byte to a machine output port.

Remarks The integer expression I is the port number. The integer expression J is the data to be transmitted.

Example

```
100 OUT 32,100
```

2.48 POKE

Format	POKE I,J
	where I and J are integer expressions.
Purpose	To write a byte into a memory location.
Remarks	I and J are integer expressions. The expression I represents the address of the memory location and J is the data byte. I must be in the range -32768 to 65535. (For interpretation of negative values of I, see "VARPTR," Section 3.43.)
	The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.28.
	POKE and PEEK are useful for storing data efficiently, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.
Example	`10 POKE &H5A00,&HFF`

2.49 PRINT

Format	PRINT [<list of expressions>]
Purpose	To output data at the terminal.

Remarks

If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. Microsoft BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, Microsoft BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8(-7) is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .0000000000000001 and 1D-16 is output as 1D-16.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X↑5
30 END
RUN
 10           0            -25            3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2
```
LIST
10 INPUT X
20 PRINT X ''SQUARED IS'' X↑2 ''AND'';
30 PRINT X ''CUBED IS'' X↑3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
 _
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line. Line 40 causes a blank line to be printed before the next prompt.

Example 3
```
10 FOR X=1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5 10 10 20 15 30 20 40 25 50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space, and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

2.50 PRINT USING

Format PRINT USING <string exp>;<list of expressions>

Purpose To print strings or numbers using a specified format.

Remarks and Examples

<list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) composed of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.

Example:

```
10 A$=''LOOK'':B$=''OUT''
30 PRINT USING ''!'';A$;B$
40 PRINT USING ''\\'';A$;B$
50 PRINT USING ''\\'';A$;B$;''!!''
RUN
LO
LOOKOUT
LOOK OUT !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output without modification.

Example:

```
10 A$=''LOOK'':B$=''OUT''
20 PRINT USING ''!'';A$;
30 PRINT USING ''&'';B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0, if necessary). Numbers are rounded as necessary.

```
PRINT USING ''##.##'';.78
 0.78
```

```
PRINT USING ''###.##'';987.654
987.65
```

```
PRINTUSING''##.##   '';10.2,5.3,66.789,.234
10.20 5.30 66.79 0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+	A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.
−	A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING ''+##.## '';-68.95,2.4,55.6,
-.9 -68.95 +2.40 +55.60 -0.90
```

```
PRINT USING ''##.##- '';-68.95,22.449,
-7.01 68.95- 22.45 7.01-
```

**	A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING ''**#.# '';12.39,-0.9,765.1
*12.4 *-0.9 765.1
```

$$	A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING ''$$###.##'';456.78
 $456.78
```

**$	The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

,

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (_ _) format.

```
PRINT USING ''####,.##'';1234.5
1,234.50

PRINT USING ''####.##,'';1234.5
1234.50,
```

↑↑↑↑

Four carets (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carets allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or -is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING ''##.##↑↑↑↑'';234.56
 2.35E+02

PRINT USING ''.####↑↑↑↑-'';888888
.8889E+06

PRINT USING ''+.##↑↑↑↑'';123
+.12E+03
```

_

An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING ''!##.##!'';12.34
!12.34!
```

The literal character itself may be an underscore by placing “ ” in the format string.

%	If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING ''##.##'';111.22
%111.22

PRINT USING ''.##'';.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

2.51 PRINT# AND PRINT# USING

Format	PRINT#<file number>,[USING <string exp>;]<list of expressions>
Purpose	To write data to a sequential cartridge file.

Remarks <file number> is the number used when the file was OPENed for output. <string exp> consists of formatting characters as described in Section 2.50, "PRINT USING." The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the cartridge. An image of the data is written to the cartridge, just as it would be displayed on the terminal screen with a PRINT statement. For this reason, care should be taken to delimit the data on the cartridge, so that it will be input correctly from the cartridge.

In the list of expressions, numeric expressions should be delimited by semicolons. For example:

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the cartridge.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the cartridge, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the cartridge. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;'',''; B$
```

The image written to cartridge is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to the cartridge surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

```
PRINT#1,A$;B$
```

would write the following image to the cartridge:

```
CAMERA, AUTOMATIC 93604-1
```

And the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly on the cartridge, write double quotation marks to the cartridge image using CHR$(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;
CHR$(34)
```

writes the following image to the cartridge:

```
''CAMERA, AUTOMATIC'''' 93604-1''
```

And the statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and "93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the cartridge file. For example:

```
PRINT#1,USING''$$###.##,'';J;K;L
```

For more examples using PRINT#, see "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*

See also "WRITE#," Section 2.68.

2.52 PUT

Format	PUT [#]<file number>[,<record number>]
Purpose	To write a record from a random buffer to a random cartridge file.
Remarks	<file number> is the number under which the file was OPENed. If <record number> is omitted, the record will assume the next available record number (after the last PUT). The largest possible record number is 32,767. The smallest record number is 1.
Example	See "Microsoft BASIC Cartridge File Handling I/O," in the *Microsoft BASIC User's Guide.*
Note	PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement.

In the case of WRITE#, Microsoft BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

2.53 RANDOMIZE

Format RANDOMIZE [<expression>]

Purpose To reseed the random number generator.

Remarks If <expression> is omitted, Microsoft BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3
        (user types 3)
 .88598 .484668 .586328 .119426 .709225
Ok
RUN
Random Number Seed (-32768 to 32767)? 4
        (user types 4 for new sequence)
 .803506 .162462 .929364 .292443 .322921
Ok
RUN
Random Number Seed (-32768 to 32767)? 3
        (same sequence as first RUN)
 .88598 .484668 .586328 .119426 .709225
Ok
```

2.54 READ

Format READ <list of variables>

Purpose To read values from a DATA statement and assign them to variables. (See "DATA," Section 2.10.)

Remarks A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an "Out of data" error message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see "RESTORE," Section 2.57)

Example 1

```
 .
 .
 .
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
 .
 .
 .
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2

```
LIST
10 PRINT ''CITY'', ''STATE'', '' ZIP''
20 READ C$,S$,Z
30 DATA ''DENVER,'', COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY          STATE         ZIP
DENVER,       COLORADO      80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

2.55 REM

Format REM <remark>

Purpose To allow explanatory remarks to be inserted in a program.

Remarks REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into from a GOTO or GOSUB statement. Execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

Important Do not use this in a data statement, because it would be considered legal data.

Example

```
 .
 .
 .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
 .
 .
 .
```

or

```
 .
 .
 .
120 FOR I=1 TO 20 'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
 .
 .
 .
```

2.56 RENUM

Format	RENUM [[<new number>][,[<old number>] [,<increment>]]]	
Purpose	To renumber program lines.	
Remarks	<new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10. RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number in xxxxx" is printed. The incorrect line number reference is not changed by RENUM, but line number yyyyy may be changed.	
Note	RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.	
Examples	RENUM	Renumbers the entire program. The first new line number will be 10. Lines will be numbered in increments of 10.
	RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will be numbered in increments of 50.
	RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and are numbered in increments of 20.

2.57 RESTORE

Format	RESTORE [<line number>]
Purpose	To allow DATA statements to be reread from a specified line.
Remarks	After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
  .
  .
  .
```

2.58 RESUME

Formats	RESUME
	RESUME 0
	RESUME NEXT
	RESUME <line number>
Purpose	To continue program execution after an error recovery procedure has been performed.

Remarks Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <line number>	Execution resumes at <line number>.

A RESUME statement that is not in an error handling routine causes a "RESUME without error" message to be printed.

Example

```
10 ON ERROR GOTO 900
 .
 .
 .
900 IF (ERR=230)AND(ERL=90) THEN PRINT
    ''TRY AGAIN'': RESUME 80
 .
 .
 .
```

2.59 RUN

Format 1 RUN [<line number>]

Purpose To execute the program currently in memory.

Remarks If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. Microsoft BASIC always returns to command level after a RUN is executed.

Example

```
RUN
```

Format 2 RUN <filename>[,R]

Purpose To load a file from cartridge into memory and run it.

Remarks <filename> is the name used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide* for information about possible filename extensions under your operating system.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example

```
RUN ''NEWFIL'',R
```

See also "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*

2.60 SAVE

Format SAVE <filename>[{,A|,P}]

Purpose To save a program file on cartridge.

Remarks <filename> is a quoted string that conforms to your operating system's requirements for filenames. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide* for information about possible filename extensions under your operating system.) If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, Microsoft BASIC saves the file in a compressed binary format. ASCII format takes more space on the cartridge, but some cartridge access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples

```
SAVE ''COM2'',A
SAVE ''PROG'',P
```

See also "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*

2.61 STOP

Format STOP

Purpose To terminate program execution and return to command level.

Remarks STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line nnnnn

Unlike the END statement, the STOP statement does not close files.

Microsoft BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.8).

Example

```
10 INPUT A,B,C
20 K=A†2*5.3:L=B†3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

2.62 SWAP

Format SWAP <variable>,<variable>

Purpose To exchange the values of two variables.

Remarks Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example

```
LIST
10 A$='' ONE '' : B$='' ALL '' : C$=''FOR''
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

2.63 TRON/TROFF

Format TRON

TROFF

Purpose To trace the execution of program statements.

Remarks As an aid in debugging, the TRON statement (executed in either direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

2.64 WAIT

Format WAIT <port number>,I[,J]

where I and J are integer expressions.

Purpose To suspend program execution while monitoring the status of a machine input port.

Remarks The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, Microsoft BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero

Important It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine. To avoid this, WAIT must have the specified value at <port number> during some point in the program execution.

Example

```
100 WAIT 32,2
```

2.65 WHILE...WEND

Format

```
WHILE <expression>
  .
  .
[<loop statements>]
  .
  .
WEND
```

Purpose

To execute a series of statements in a loop as long as a given condition is true.

Remarks

If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. Microsoft BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example

```
 90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115         FLIPS=0
120         FOR I=1 TO J-1
130                   IF A$(I)>A$(I+1) THEN
                            SWAP A$(I),
                            A$(I+1):FLIPS=1
140         NEXT I
150 WEND
```

2.66 WIDTH

Format	WIDTH [LPRINT]<integer expression>
Purpose	To set the printed line width in number of characters for the terminal or line printer.

Remarks

If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

<integer expression> must have a value in the range 15 to 255. The default width is 72 characters.

If <integer expression> is 255, the line width is "infinite"; that is, Microsoft BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

Example

```
10 PRINT ''ABCDEFGHIJKLMNOPQRSTUVWXYZ''
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

2.67 WRITE

Format WRITE [<list of expressions>]

Purpose To output data at the terminal.

Remarks If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions. They must be separated by commas.

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, Microsoft BASIC inserts a carriage return/ line feed.

WRITE outputs numeric values using the same format as the PRINT statement. (See Section 2.49.)

Example

```
10 A=80:B=90:C$=''THAT'S ALL''
20 WRITE A,B,C$
RUN
 80, 90,''THAT'S ALL''
Ok
```

2.68 WRITE#

Format WRITE#<file number>,<list of expressions>

Purpose To write data to a sequential file.

Remarks <file number> is the number under which the file was OPENed in "O" mode (see "OPEN," Section 2.45). The expressions in the list are string or numeric expressions. They must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to cartridge and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to cartridge.

Example Let `A$=''CAMERA''` and `B$=''93604-1''`

The statement:

```
WRITE#1,A$,B$
```

writes the following image to cartridge:

"CAMERA","93604-1"

A subsequent INPUT# statement, such as

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

Chapter 3 Microsoft BASIC Functions

Microsoft BASIC intrinsic functions are described in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions.
I and J	Represent integer expressions.
X$ and Y$	Represent string expressions.

If a floating-point value is supplied where an integer is required, Microsoft BASIC will round the fractional portion and use the resulting integer.

3.1 ABS

Format ABS(X)

Format ABS(X)

Action Returns the absolute value of the expression X.

Example

```
PRINT ABS(7*(-5))
 35
Ok
```

3.2 ASC

Format ASC(X$)

Action Returns a numerical value that is the ASCII code for the first character of the string X$. (See Appendix C for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

Example

```
10 X$=''TEST''
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function, Section 3.5, for details on ASCII-to-string conversion.

3.3 ATN

Format ATN(X)

Action Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
Ok
```

3.4 CDBL

Format CDBL(X)

Action Converts X to a double precision number.

Example

```
10 A=454.67
20 PRINT A;CDBL(A)
RUN
 454.67 454.6700134277344
Ok
```

3.5 CHR$

Format CHR$(I)

Action Returns a string whose one character is ASCII character I. (ASCII codes are listed in Appendix C.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character (CHR$(7)) could be sent as a preface to an error message, or a form feed (CHR$(12)) could be sent to clear a terminal screen and return the cursor to the home position.

Example
```
PRINT CHR$(66) B
Ok
```

See the ASC function, Section 3.2, for details on ASCII-to-numeric conversion.

3.6 CINT

Format CINT(X)

Action Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Example
```
PRINT CINT(45.67)
 46 Ok
```

See the CDBL and CSNG functions for details on converting numbers to the double precision and single precision data type, respectively. See also the FIX and INT functions, both of which return integers.

3.7 COS

Format COS(X)

Action Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example

```
10 X=2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

3.8 CSNG

Format CSNG(X)

Action Converts X to a single precision number.

Example

```
10 A# = 975.3421# 20 PRINT A#; CSNG(A#) RUN
 975.3421 975.342 Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types, respectively.

3.9 CVI, CVS, CVD

Format CVI(<2-byte string>) CVS(<4-byte string>) CVD(<8-byte string>)

Action Convert string values to numeric values. Numeric values that are read in from a random cartridge file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ... 80 GET #1 90
Y=CVS(N$)
.
.
.
```

See also "MKI$, MKS$, MKD$," Section 3.26 and "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*

3.10 EOF

Format EOF(<file number>)

Action Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid "Input past end" errors.

Example

```
10 OPEN ''I'',1,''DATA''
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
.
.
.
```

3.11 EXP

Format EXP(X)

Action Returns e (base of natural logarithms) to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example

```
10 X=5
20 PRINT EXP(X-1)
RUN
 54.5982
Ok
```

3.12 FIX

Format FIX(X)

Action Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)* INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Examples PRINT FIX(58.75)

```
 58
Ok
```

```
PRINT FIX(-58.75)
-58
Ok
```

3.13 FRE

Format FRE(0) FRE(" ")

Action Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by Microsoft BASIC.

FRE(" ") forces a garbage collection before returning the number of free bytes. Be patient: garbage collection may take 1 to 1-1/2 minutes.

Microsoft BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE(" ") periodically will result in shorter delays for each garbage collection.

Example

```
PRINT FRE(0)
 14542
Ok
```

3.14 HEX$

Format HEX$(X)

Action Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated.

Example

```
10 INPUT X
20 A$=HEX$(X)
30 PRINT X ''DECIMAL IS '' A$ '' HEXADECIMAL''
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ function, Section 3.27, for details on octal conversion.

3.15 INKEY$

Format INKEY$

Action Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal. No characters will be echoed. All characters are passed through to the program except for CMD-C, which terminates the program.

Example

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$=''''
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

3.16 INP

Format INP(I)

Action Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement, Section 2.47.

Example

```
100 A=INP(255)
```

3.17 INPUT$

Format INPUT$(X[,[#]Y])

Action Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed. All control characters are passed through except CMD-C, which is used to interrupt the execution of the INPUT$ function.

Example 1

```
 5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
   HEXADECIMAL
10 OPEN''I'',1,''DATA''
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2

```
 .
 .
 .
100 PRINT ''TYPE P TO PROCEED OR S TO STOP''
110 X$=INPUT$(1)
120 IF X$=''P'' THEN 500
130 IF X$=''S'' THEN 700 ELSE 100
 .
 .
 .
```

3.18 INSTR

Format INSTR([I,]X$,Y$)

Action Searches for the first occurrence of string Y$ in X$, and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 1 to 255. If I is greater than the number of characters in X$ (LEN(X$)), or if X$ is null or Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions, or string literals.

Example

```
10 X$=''ABCDEB''
20 Y$=''B''
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2 6
Ok
```

Note If I=0 is specified, the "Illegal function call" error message will be returned.

3.19 INT

Format INT(X)

Action Returns the largest integer <=X.

Examples

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

See the CINT and FIX functions, Sections 3.6 and 3.12, respectively, which also return integer values.

3.20 LEFT$

Format LEFT$(X$,I)

Action Returns a string comprising the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than the number of characters in X$ (LEN(X$)), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Example

```
10 A$=''BASIC''
20 B$=LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Also see the MID$ and RIGHT$ functions, Sections 3.25 and 3.30, respectively.

3.21 LEN

Format LEN(X$)

Action Returns the number of characters in X$. Nonprinting characters and blanks are counted.

Example

```
10 X$=''PORTLAND, OREGON''
20 PRINT LEN(X$)
 16
Ok
```

3.22 LOC

Format	LOC(<file number>) where <file number> is the number under which the file was OPENed.
Action	With random cartridge files, LOC returns the record number just read or written from a GET or PUT statement. If the file was opened but no cartridge I/O has been performed yet, LOC returns a 0. With sequential files, LOC returns the number of sectors (128-byte blocks) read from or written to the file since it was OPENed.
Example	`200 IF LOC(1)>50 THEN STOP`

3.23 LOG

Format	LOG(X)
Action	Returns the natural logarithm of X. X must be greater than zero.

Example

```
PRINT LOG(45/7)
 1.86075 Ok
```

3.24 LPOS

Format	LPOS(X)
Action	Returns the current position of the line printer print head within the line printer's buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

Example

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

3.25 MID$

Format MID$(X$,I[,J])

Action Returns a string of length J characters from X$, beginning with the Ith character. I and J must be in the range 1 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I is greater than the number of characters in X$ (LEN(X$)), MID$ returns a null string.

Example

```
LIST
10 A$=''GOOD ''
20 B$=''MORNING EVENING AFTERNOON''
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

Also see the LEFT$ and RIGHT$ functions, Sections 3.20 and 3.30, respectively.

If I=0 is specified, the "Illegal function call" error message will be returned.

3.26 MKI$, MKS$, MKD$

Format MKI$(<integer expression>) MKS$(<single precision expression>) MKD$ (<double precision expression>)

Action Convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example

```
 90 AMT=(K+T)
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$= MKS$(AMT)
120 LSET N$=A$
130 PUT #1
  .
  .
```

See also "CVI, CVS, CVD," Section 3.9 and "Microsoft BASIC Cartridge File Handling," in the *Microsoft BASIC User's Guide.*

3.27 OCT$

Format OCT$(X)

Action Returns a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example

```
PRINT OCT$(24)
 30
Ok
```

See the HEX$ function, Section 3.14, for details on hexadecimal conversion.

3.28 PEEK

Format	PEEK(I)
Action	Returns the byte read from the indicated memory location (I).
Remarks	The returned value is an integer in the range 0 to 255. I must be in the range -32768 to 65535. (For the interpretation of a negative value of I, see "VARPTR," Section 3.43.) PEEK is the complementary function of the POKE statement.
Example	`A=PEEK(&H5A00)`

3.29 POS

Format	POS(I)
Action	Returns the current cursor position. The leftmost position is 1. X is a dummy argument.
Example	`IF POS(X)>60 THEN PRINT CHR$(13)` Also see the LPOS function, Section 3.24.

3.30 RIGHT$

Format RIGHT$(X$,I)

Action Returns the rightmost I characters of string X$. If I is equal to the number of characters in X$ (LEN(X$)), returns X$. If I=0, the null string (length zero) is returned.

Example

```
10 A$=''DISK BASIC''
20 PRINT RIGHT$(A$,5)
RUN
BASIC
Ok
```

Also see the LEFT$ and MID$ functions, Sections 3.20 and 3.25, respectively.

3.31 RND

Format RND[(X)]

Action Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see "RANDOMIZE," Section 2.53). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24 30 31 51 5
Ok
```

Note The values produced by the RND function may vary with different implementations of Microsoft BASIC.

3.32 SGN

Format SGN(X)

Action If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example
```
ON SGN(X)+2 GOTO 100,200,300
```

branches to 100 if X is negative, 200 if X is 0, and 300 if X is positive.

3.33 SIN

Format SIN(X)

Action Returns the sine of X in radians. SIN(X) is calculated in single precision. COS(X)=SIN(X+3.14159/2).

Example
```
PRINT SIN(1.5)
 .997495
Ok
```

See also the COS(X) function, Section 3.7.

3.34 SPACE$

Format SPACE$(X)

Action Returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

Example
```
10 FOR I=1 TO 5
20 X$=SPACE$(I)
30 PRINT X$;I 40 NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

Also see the SPC function, Section 3.35.

3.35 SPC

Format SPC(I)

Action Prints I blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255. A';' is assumed to follow the SPC(I) command.

Example
```
PRINT ''OVER'' SPC(15) ''THERE''
OVER                THERE
Ok
```

Also see the SPACE$ function, Section 3.34.

3.36 SQR

Format SQR(X)

Action Returns the square root of X. X must be >=0.

Example

```
10 FOR X=10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10          3.16228
 15          3.87298
 20          4.47214
 25          5
Ok
```

3.37 STR$

Format STR$(X)

Action Returns a string representation of the value of X.

Example

```
 5 REM ARITHMETIC FOR KIDS
10 INPUT ''TYPE A NUMBER'';N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,
   400,500
 .
 .
 .
```

Also see the VAL function, Section 3.42.

3.38 STRING$

Formats STRING$(I,J)
STRING$(I,X$)

Action Returns a string of length I whose characters all have ASCII code J or the first character of X$.

Example

```
10 X$=STRING$(10,45)
20 PRINT X$ ''MONTHLY REPORT'' X$
RUN
----------MONTHLY REPORT----------
Ok
```

3.39 TAB

Format TAB(I)

Action Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example

```
10 PRINT ''NAME'' TAB(25) ''AMOUNT'' :
   PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA ''G. T. JONES'',''$25.00''
RUN
NAME                    AMOUNT

G. T. JONES             $25.00
Ok
```

3.40 TAN

Format TAN(X)

Action Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example

```
10 Y=Q*TAN(X)/2
```

3.41 USR

Format USR[<digit>](X)

Action Calls the user's assembly language subroutine with the argument X. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See "Assembly Language Subroutines," in the *Microsoft BASIC User's Guide.*

Example

```
40 B=T*SIN(Y)
50 C=USR(B/2)
60 D=USR(B/3)
  .
  .
  .
```

3.42 VAL

Format VAL(X$)

Action Returns the numerical value of string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example,

VAL(" -3")

returns -3.

Example

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL (ZIP$)<90000 OR VAL(ZIP$)>96699
     THEN PRINT NAME$ TAB(25) ''OUT OF STATE''
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=
   90815 THEN PRINT NAME$ TAB(25) ''LONG
   BEACH''
  .
  .
  .
```

See the STR$ function, Section 3.37, for details on numeric-to-string conversion.

3.43 VARPTR

Format 1 VARPTR(<variable name>)

Format 2 VARPTR(#<file number>)

Action Format 1

Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array). For string variables, the address of the first byte of the string descriptor is returned (see "BASIC Assembly Language Subroutines," in the *Microsoft BASIC User's Guide* for discussion of the string descriptor). The address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

Note All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Format 2

For sequential files, returns the starting address of the cartridge I/O buffer assigned to <file number>. For random files, returns the address of the FIELD buffer assigned to <file number>.

Example

```
100 X=USR(VARPTR(Y))
```

Appendices

APPENDIX A

Error Codes and Error Messages

Code	No.	Message
NF	1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
SN	2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.). Microsoft BASIC automatically enters edit mode at the line that caused the error.
RG	3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
OD	4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
FC	5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 1. A negative or unreasonably large subscript. 2. A negative or zero argument with LOG. 3. A negative argument to SQR. 4. A negative mantissa with a noninteger exponent.

5. A call to a USR function for which the starting address has not yet been given.

6. An improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON... GOTO.

OV 6 Overflow

The result of a calculation is too large to be represented in Microsoft BASIC number format. If underflow occurs, the result is zero and execution continues without an error.

OM 7 Out of memory

A program is too large, or has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

UL 8 Undefined line

A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.

BS 9 Subscript out of range

An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.

DD 10 Redimensioned array

Two DIM statements are given for the same array; or, a DIM statement is given for an array after the default dimension of 10 has been established for that array.

/0 11 Division by zero

A division by zero is encountered in an expression; or, the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

ID 12 Illegal direct

A statement that is illegal in direct mode is entered as a direct mode command.

TM 13 Type mismatch

A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

OS 14 Out of string space

String variables have caused BASIC to exceed the amount of free memory remaining. Microsoft BASIC will allocate string space dynamically, until it runs out of memory.

LS 15 String too long

An attempt is made to create a string more than 255 characters long.

ST 16 String formula too complex

A string expression is too long or too complex. The expression should be broken into smaller expressions.

CN 17 Can't continue

An attempt is made to continue a program that:

1. Has halted due to an error.

2. Has been modified during a break in execution.

3. Does not exist.

UF 18 Undefined user function

A USR function is called before the function definition (DEF statement) is given.

19 No RESUME

An error handling routine is entered but contains no RESUME statement.

20 RESUME without error

A RESUME statement is encountered before an error handling routine is entered.

21 Unprintable error

An error message is not available for the error condition which exists.

22 Missing operand

An expression contains an operator with no operand following it.

23 Line buffer overflow

An attempt has been made to input a line that has too many characters.

26 FOR without NEXT

A FOR statement was encountered without a matching NEXT.

29 WHILE without WEND

A WHILE statement does not have a matching WEND.

30 WEND without WHILE

A WEND statement was encountered without a matching WHILE.

Cartridge Errors

50 Field overflow

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error

An internal malfunction has occurred in Microsoft BASIC. Report to Microsoft the conditions under which the message appeared.

52 Bad file number

A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found

A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode

An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN statement with a file mode other than I, O, or R.

55 File already open

A sequential output mode OPEN statement is issued for a file that is already open; or a KILL statement is given for a file that is open.

57	Disk I/O error
	An I/O error occurred on a cartridge I/O operation. It is a fatal error; i.e., the operating system cannot recover from the error.
58	File already exists
	The filename specified in a NAME statement is identical to a filename already in use on the cartridge.
61	Disk full
	All cartridge storage space is in use.
62	Input past end
	An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.
63	Bad record number
	In a PUT or GET statement, the record number is either greater than the maximum allowed (32,767) or equal to zero.
64	Bad file name
	An illegal form is used for the filename with a LOAD, SAVE, KILL, or OPEN statement (e.g., a filename with too many characters).
66	Direct statement in file
	A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.
67	Too many files
	An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

APPENDIX B

Mathematical Functions

Derived Functions

Functions that are not intrinsic to Microsoft BASIC may be calculated as follows.

Function	Microsoft BASIC Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X +1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR(-X*X +1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=(EXP(X)-EXP(-X))/ (EXP(X)+EXP(-X))
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=(EXP(X)+EXP(-X))/ (EXP(X)-EXP(-X))
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X +1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X -1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/ (1-X))/2

INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X +1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X) *SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/ (X-1))/2

APPENDIX C

ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	—
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	↑
009	09H	HT	052	34H	4	095	5FH	_
010	0AH	LF	053	35H	5	096	60H	‘
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	”	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z

APPENDIX C

ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
038	26H	&	081	51H	Q	124	7CH	|
039	27H	‘	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	..
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal, Hex=hexadecimal (H), CHR=character, LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

APPENDIX D

MICROSOFT BASIC RESERVED WORDS

The following is a list of reserved words used in Microsoft BASIC.

ABS
AND
ASC
ATN
AUTO
CALL
CDBL
CHAIN
CHR$
CINT
CLEAR
CLOSE
COMMON
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DEFDBL
DEFINT
DEFSNG
DEFSTR
DEF FN
DEF USR
DELETE
DIM
EDIT
ELSE
END
EOF

ERASE
ERL
ERR
ERROR
END
EXP
FIELD
FILES
FIX
FOR
FRE
GET
GOSUB
HEX$
IF
IMP
INP
INPUT
INKEY$
INPUT#
INPUT$
INSTR
INT
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC

LOF
LOG
LPOS
LPRINT
LSET
MERGE
MID$
MKD
MKI$
MKS$
MOD
NAME
NEW
NOT
OCT$
ON
OPENON
OPTION
OR
PEEK
POKE
POS
PPRINT
PRINT# USING VAL
PUT
RANDOMIZE WAIT
READ
REM
RENUM
RESET
RESTORE
RESUME

RIGHT$
RND
RSET
RUN
SAVE
SBN
SIN
SPACE
SPC
SQR
STOP
STR$
STRING$
SWAP
SYSTEM
TAB
TAN
THEN
TO
TROFF
TRON

USR

VARPTR

WEND
WHILE
WRITE
WRITE#
XOR

BEAVERITE PRODUCTS INCORPORATED ®

BEAVER FALLS, N.Y.
MADE IN U.S.A.

VINYL MEETS SPECIFICATIONS
FOR TGS/LLD CLASS IN
N 78778
1 84